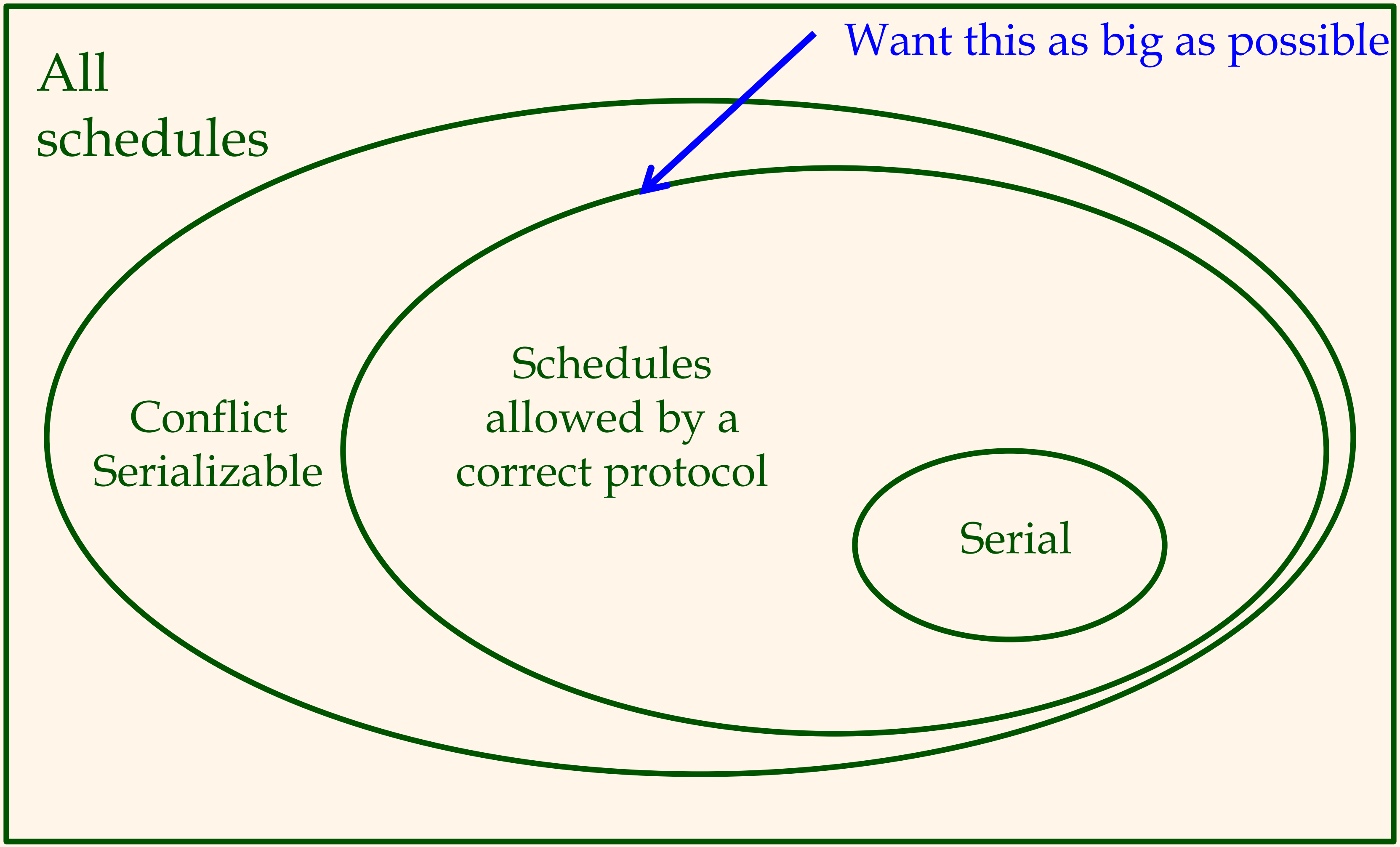
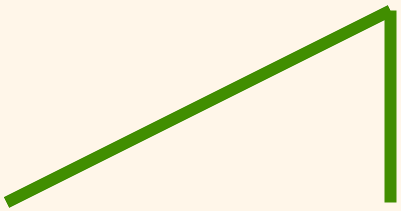
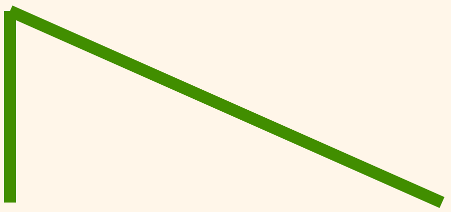
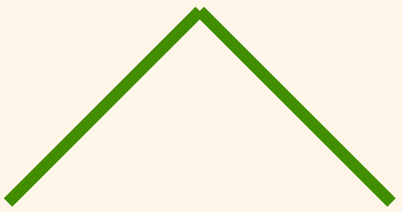
Locking continued

Concurrency control protocols



Last time: 2PL

Conservative Strict	Yes	No
Yes		
No		

A final theoretical note on 2PL

- Even non-strict, non-conservative 2PL is "too much" (forbids some interleavings that are ok)
  - Every interleaving/schedule permitted by 2PL is conflict-serializable
  - But not necessarily vice versa!
  - W1(A) R2(A) Commit2 R3(B) Commit3 W1(B) Commit1

In practice:

- Strict 2PL is the most commonly used locking protocol
  - Locks acquired as needed
  - Locks held until commit
- Means deadlock is an issue and must be dealt with

Deadlocks

- Deadlock: Cycle of transactions waiting for locks to be released by each other.
- Two ways of dealing with deadlocks:
  - Deadlock prevention
  - Deadlock detection

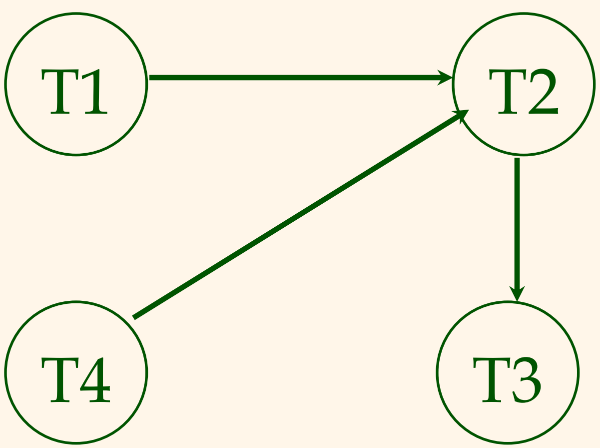
Deadlock Detection

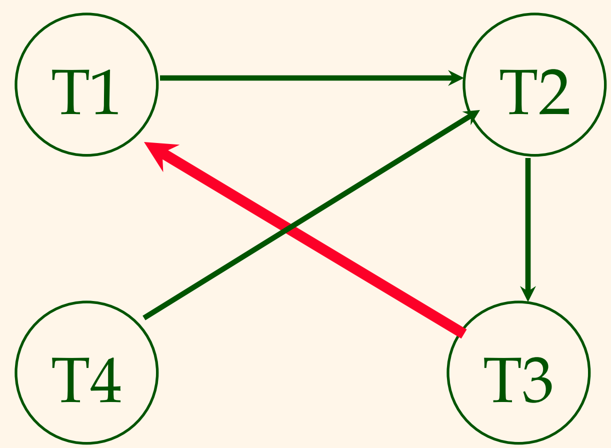
- Create a waits-for graph:
  - Nodes are transactions
  - There is an edge from Ti to Tj if Ti is waiting for Tj to release a lock
- Periodically check for cycles in the waits-for graph
- If cycle found, abort one of the transactions (its locks get released allowing others to proceed)

Deadlock Detection

Example: (S = request shared lock, X = request exclusive lock)

T1:	S(A), R(A),		S(B)				
T2:		X(B),W(B)			X(C)		
T3:				S(C), R(C)			X(A)
T4:						X(B)	





Deadlock Prevention

- Assign priorities based on timestamps.
  - Lower timestamp (older transaction) => higher priority
- Assume Ti wants a lock that Tj holds. Two policies are possible:
  - Wound-wait: If Ti has higher priority, Tj aborts; otherwise Ti waits
  - Wait-die: It Ti has higher priority, Ti waits for Tj; otherwise Ti aborts
- If a transaction re-starts, make sure it has its original timestamp to avoid starvation (note the higher-priority transaction is never aborted in either scheme)

Deadlock Prevention

- Advantage of wait-die:
  - Non-preemptive (if I have all my locks will never be aborted for deadlock reasons)
- Disadvantage of wait-die:
  - A younger transaction that conflicts with an older transaction may be repeatedly aborted for the same reason

Next

- So far have assumed
  - all operations are on a single object
  - all objects are independent
  - no objects are created or deleted
- If these assumptions fail
  - this may be bad - problems may arise
  - or good - there may be opportunities for more efficient locking protocols.

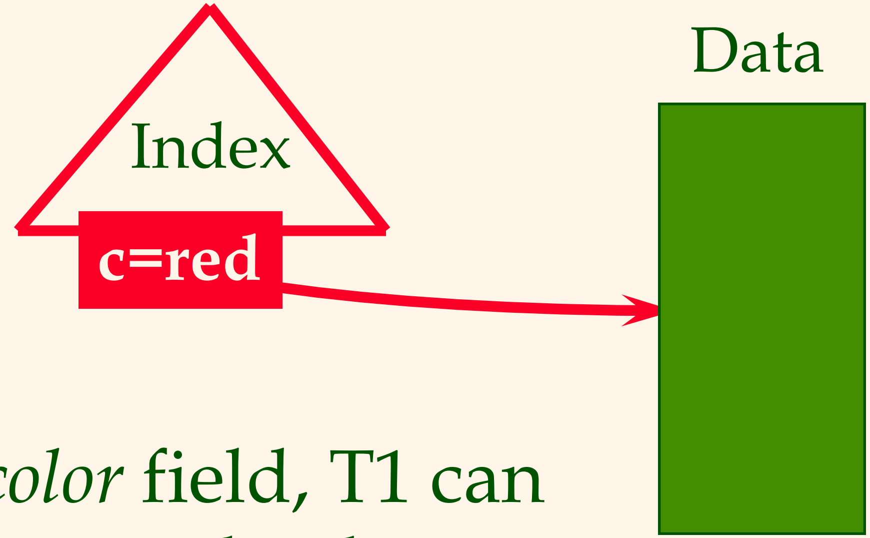
The Phantom Problem

- Suppose we have Strict 2PL and transaction 1 performs the following query **twice:**
  - `SELECT * FROM Widgets WHERE color = 'red';`
- In the interim, transaction 2 inserts a new red widget
  - Can do this – T1 only has locks on pre-existing red widgets!
- Loss of isolation even though we used 2PL!

Avoiding phantoms

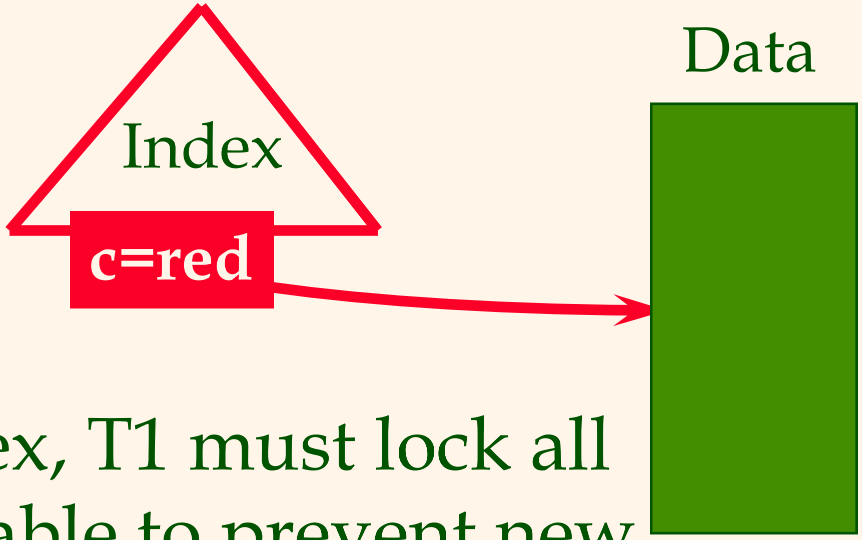
- The problem is not that 2PL is incorrect, the problem is that T1 did not "really" lock all red widgets
- Need way to lock a set of tuples by specifying all possible tuples that could be in the set
  - Predicate locking
  - Can be computationally expensive with arbitrary predicates
  - In practice, could be done with index locking

Index Locking



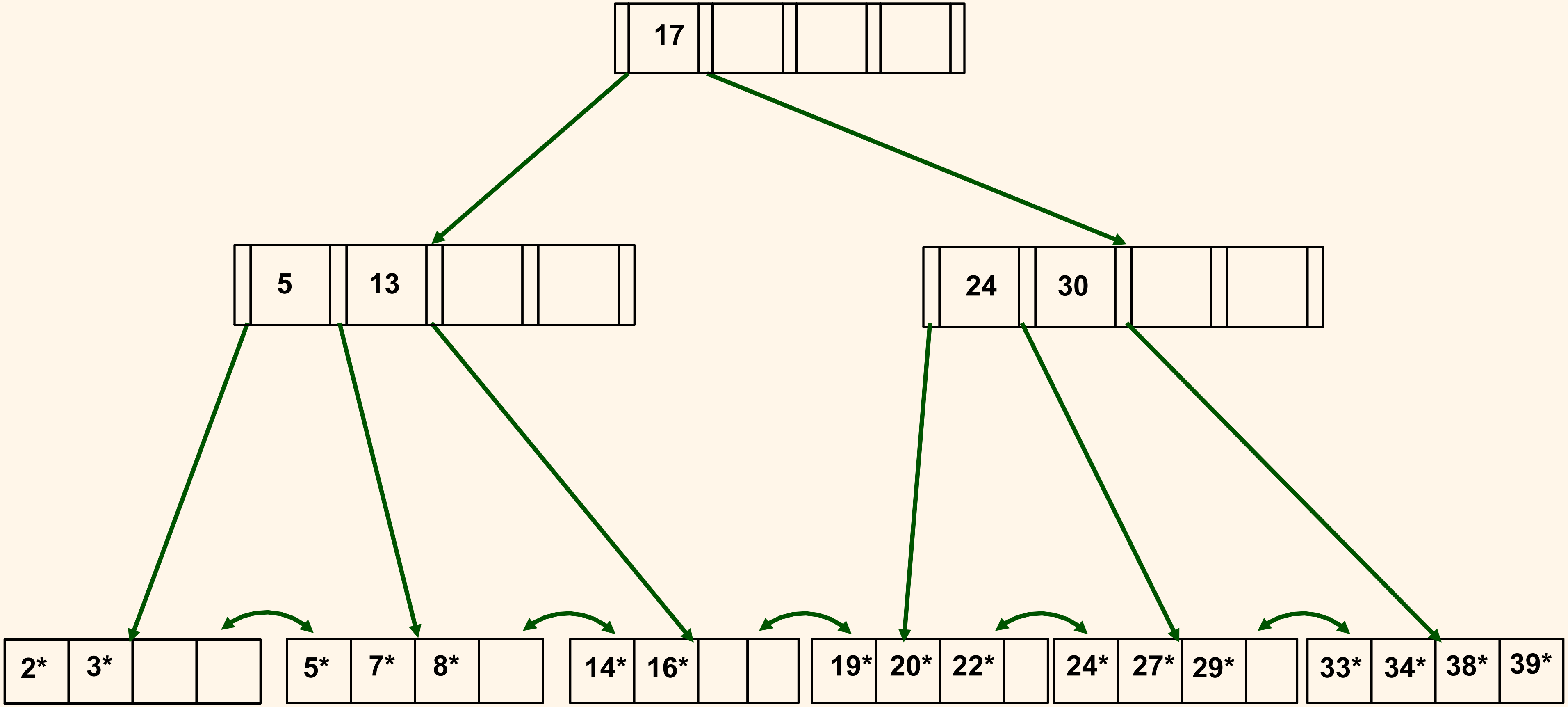
- If there is a index on the *color* field, T1 can lock the index page containing the data entries with *color* = red.
  - If there are no entries with *color* = red, T1 must lock the index page where such a data entry *would* be, if it existed!

Index Locking



- If there is no suitable index, T1 must lock all pages, and lock the file/table to prevent new pages from being added, to ensure that no new records with *color* = red are added.

Locking in B+ trees



Locking in B+ Trees

- How can we enable "safe" concurrent access to index structures?
- Note: this is useful even apart from index locking
  - If you have multiple transactions that need to search/insert/delete the tree

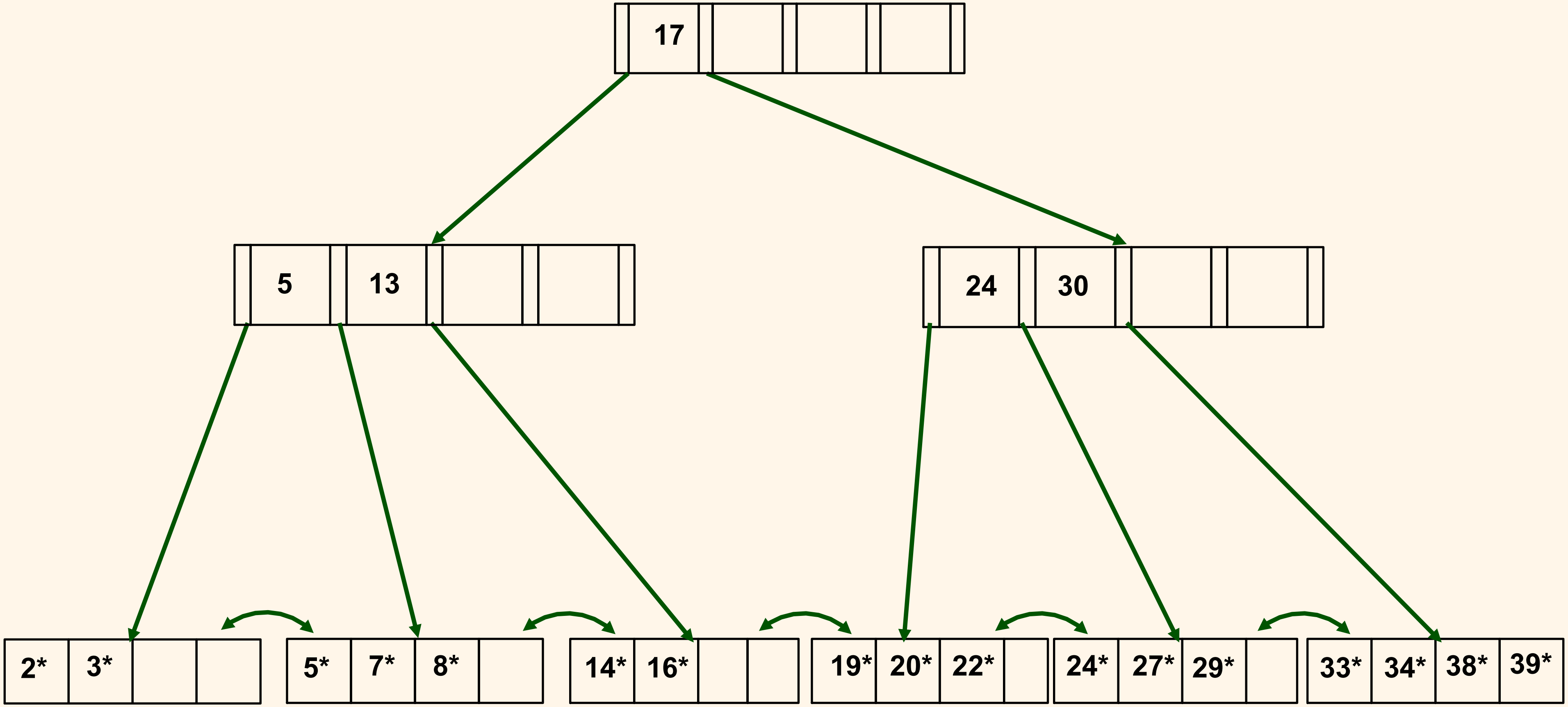
Locking in B+ Trees

- One solution: Ignore the tree structure, just lock pages while traversing the tree, following 2PL (objects to be locked = pages).
- Problem:
  - This has terrible performance!
  - Root node (and nodes close to root) become bottlenecks because every tree access begins at root.

Useful Observations

- Higher levels of the tree only direct searches for leaf pages.
  - Searches never go back to the root
  - So could release lock on root (and other nodes close to the root) early
  - OK even if we violate 2PL

Locking in B+ trees



Two Useful Observations

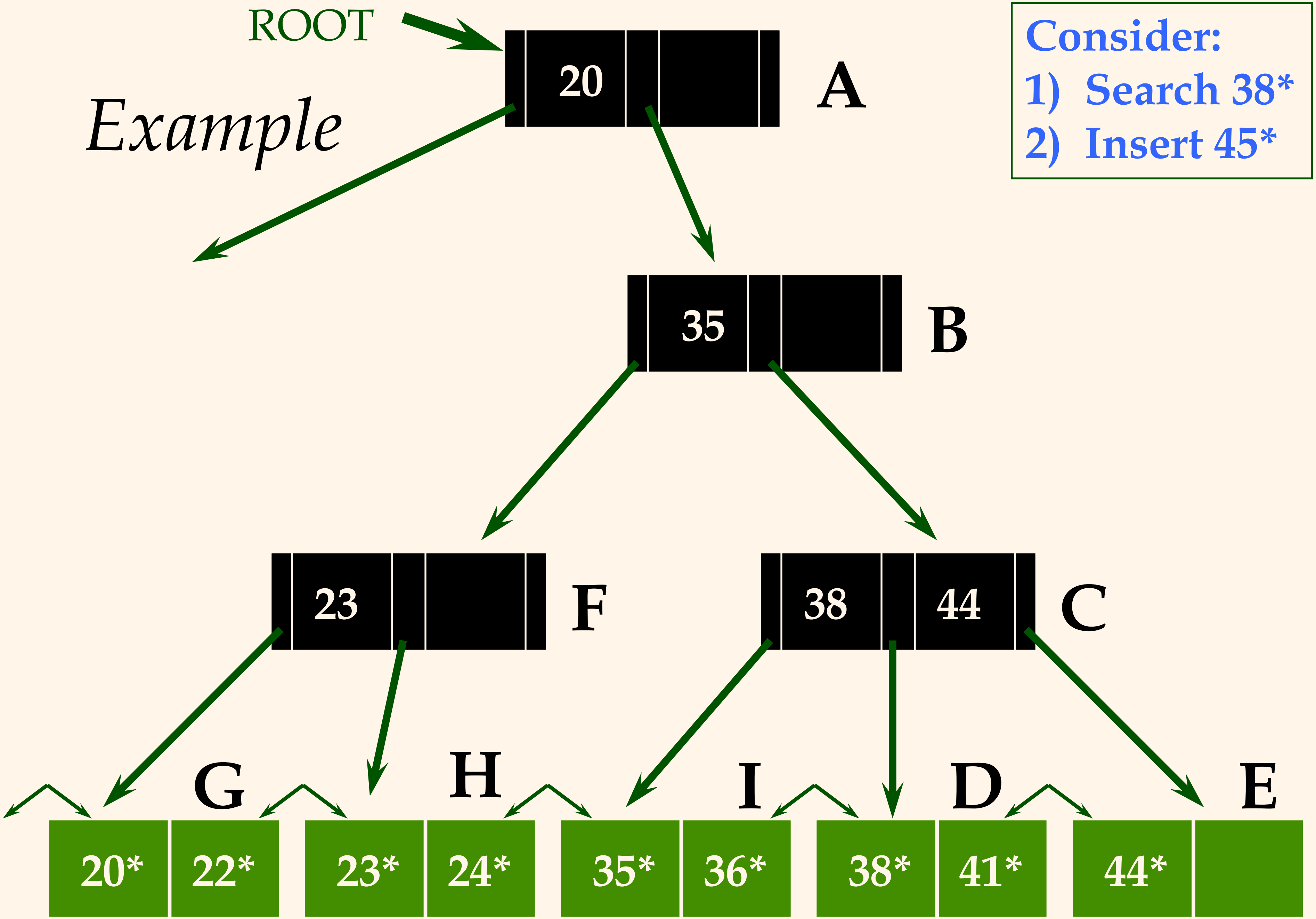
- For inserts, a node on a path from root to modified leaf must be locked (in X mode, of course), only if a split can propagate up to it from the modified leaf. (Similar point holds w.r.t. deletes.)
- So could also unlock some nodes early

A Simple Tree Locking Algorithm

- Search: Start at root and go down; repeatedly, S lock child then unlock parent.

Example

Consider:
1) **Search 38***
2) **Insert 45***



A Simple Tree Locking Algorithm

- Insert/Delete: Start at root and go down, obtaining X locks as needed. Once child is locked, check if it is safe:
  - If child is safe, release all locks on ancestors.
- Safe node: Node such that changes will not propagate up beyond this node.
  - Inserts: Node is not full.
  - Deletes: Node is not half-empty.

Improvements

- Insert/Delete sets a lot of exclusive locks which may not be necessary especially at the top levels
- Refinements based on requesting S locks until get to the leaf, then locking leaf only in X mode
  - Gambles that leaf won't need to split (common case)
  - If we are unlucky and leaf does split, need to handle it somehow
    - E.g. release all locks and restart using previous algorithm
    - Or try to request lock *upgrades* from S to X

Multiple-Granularity Locks

- Suppose we have a hierarchy on data "containers"
- Can lock at different levels